Quality Events: A Flexible Mechanism for Quality of Service Management

Richard West
Computer Science Department
Boston University
Boston, MA 02215
richwest@cs.bu.edu

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
schwan@cc.gatech.edu

Abstract

This paper describes a software mechanism, called quality events*, that utilizes application- and/or system-level service extensions to provide quality of service (QoS) guarantees to end users. Such extensions offer flexibility in: (1)* `how` *application- and/or system-level services are dynamically managed to maintain required quality, (2)* `when` *such management occurs, and (3)* `where` *this service management is performed.*

Several adaptive QoS management strategies are implemented with quality events and compared with respect to their ability to meet application-specific QoS requirements. These management strategies have different service adaptation latencies, and different degrees of coordination between services. Significant performance variations observed for these alternative strategies demonstrate the importance of a flexible QoS management mechanism like quality events. Finally, we show that adaptive QoS and, hence, resource management strategies can lead to more efficient use of resources, and better qualities of service for certain applications than non-adaptive resource management methods.

1. Introduction

Providing QoS guarantees to real-time and multimedia applications is complicated by runtime variations in resource requirements and availability. For example, an application may require more or less CPU cycles to identify a target object in a graphical image, depending on the content of that image. Likewise, resource availability may change due to a dynamically changing number of application tasks sharing a finite set of common resources. Consequently, reservation-based resource management techniques, like RSVP for network communications, and others for CPU usage [16, 13], have been developed, to provide enough resources to guarantee qualities of service even when there are variations in resource demands and availability. However, by combining these resource reservation techniques with runtime (or dynamic) quality management methods, that monitor and adapt applications and system services, it is possible to improve the quality of service to applications, whenever additional resources become available.

Several advantages are derived from dynamic QoS management via runtime adaptation. First, by reserving only the minimum amount of each resource needed by each application, overall system utilization and scalability may be improved, especially when applications exhibit runtime variations in resource requirements [9, 20]. Second, by dealing with dynamic changes in resource availability, adaptive methods can sometimes improve the overall quality experienced by end users [1, 15, 14, 18]. In response to these advantages, researchers have created infrastructures for dynamic program adaptation [4], toolkits for creating and deploying powerful adaptation techniques [19, 8], and QoS architectures [17, 2] designed for multimedia video and audio applications. Other related work has focused on application-level *specification* of quality requirements [7, 10], *translation* of quality requirements at the various stages of the system [6, 10] , and *evaluation* of the utility of a given quality of service to an application, using reward [1], value [12], or payoff [14] functions.

This paper is concerned with the effective *deployment* of adaptive methods for performing quality management, for individual and/or classes of applications and services. We describe a novel mechanism for deploying desired quality management methods and techniques, termed *quality events*. Quality events are an efficient means for developers to create flexible quality management methods that are easily varied in: (1) `how` application- or system-level services are adapted to maintain desired levels of quality, (2) `when` such adaptations occur, and (3) `where` these adaptations are performed. Specifically, using quality events, an application and/or system may be extended [3], at runtime, to enable quality management. This is done by associating with system- or application-level service providers, application-specific functions that produce, consume, and act on events, thereby performing appropriate monitoring and management tasks. For example, an application-specific function may extend a system-level service provider to affect the way in which resource allocation is performed on its behalf, or to simply inform the application about system-level resource demand. In addition, an application-level 'handler' function may adjust the application's resource requirements in response to monitored changes in resource availability. Furthermore, an application may simultaneously deploy multiple handlers, events, and channels to target multiple system services and application components, thereby creating distributed and coordinated dynamic quality management strategies. The resulting flexibility of runtime adaptation implemented with quality events is a strong attribute of our work, one that is not easily realized with other quality management infrastructures (e.g., like those described in [9, 22]).

As part of the quality event mechanism, applications specify: (1) *monitor* functions that are written to capture service quality at specified times and generate events if service adaptation is required, and (2) *handler* functions, that are executed in response to adaptation events raised by service providers. As a result, applications control which services are adapted by specifying which service providers (or *managers*) execute monitor functions and which service providers execute the corresponding management handlers. Toward these ends, *event channels* are established between monitors in specific service providers and their corresponding management handlers, which may reside in the same or other service providers. In this fashion, services are provided in a manner that is specific to the needs of individual applications. Applications can monitor and detect resource bottlenecks, adapt their requirements for heavily-demanded resources, or adapt to different requirements of alternative resources, in order to improve or maintain their quality of service.

Contributions. This paper shows how quality events can be used to construct various approaches to QoS management, thereby emphasizing the flexibility of quality events. We compare several alternative QoS management strategies, that differ in the manner in which services are coordinated and adapted to meet the QoS requirements of applications. For instance, we show for a client-server video application that feedback-based, 'upstream' management methods can be outperformed by strategies where adaptations occur as application-level data is being processed and/or transferred, 'downstream' along the logical path of resources leading to the destination. Finally, we show that adaptive resource management strategies can lead to more efficient use of resources, and better qualities of service for certain applications than non-adaptive resource management methods.

Overview. The next section describes quality events and how they are used in a QoS infrastructure, called Dionisys. Section 3 describes issues and trade-offs in the implementation of different adaptive quality management strategies. Section 4 presents an experimental evaluation of quality events, using a client-server multimedia application. Finally conclusions and future work are discussed in Section 5.

2. Quality Events: Definition and Use

Quality Event Definition. The quality event mechanism enables flexible quality management strategies to be implemented. A quality event is generated by a *monitor* function (i.e., a *producer*) in response to an observed quality of service provided to a specific application, that is either unacceptable or less than desired. In response to the generation of an event, an application-specific *handler* function (i.e., a *consumer*) provides 'hints' to a service provider that a change in service is required. Quality events pass from application- or system-level service providers, where service is monitored, to other application- or system-level service providers, where service changes are enacted. An *event channel* is established between a monitor in one service provider and one or more corresponding handlers, which may reside in the same or other service providers. Observe that we refer to an *event* as an entity that is passed along an event channel, but collectively, event channels, producers, consumers and events all form the mechanism that we call *quality events*.

Stated concisely, the quality event mechanism comprises:

- **Service Managers (SMs)** that provide runtime-adjustable service allocation policies, while also executing the monitoring and handling functions;
- **Monitor functions**, which trigger potential service adaptations, if the actual quality of a given service type is unacceptable, or if it is less than desired;
- **Handler functions**, which influence *how* service and, hence, the allocation of resources among competing applications is adapted;
- **Quality Events** generated by the execution of a monitor function *when* one or more service types are required to be adapted. The attributes associated with an event are used to decide the extent to which service should be adapted in the target service manager; and
- **Event Channels**, which allow quality events to be communicated from monitor to handler functions, that execute in the contexts of different system-level service managers or application-level processes, *where* service should be adapted.

Event channels are capable of linking monitor and handler functions that reside within the same address space, or

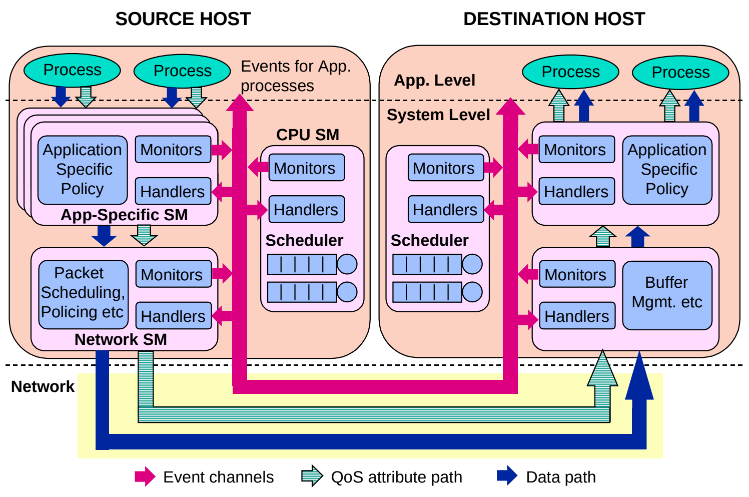

Figure 1. QoS infrastructure using quality events: The Dionisys example.

in different address spaces on the same or different hosts. Consequently, service managers are able to monitor and adapt their own service, or that of other service managers running in potentially different address spaces. The rate at which service adjustments are made is determined primarily by the rate at which service managers execute their monitor and handler functions.

Using Quality Events – The Dionisys Example. Using quality events, we have implemented a QoS infrastructure called Dionisys (see Figure 1), as a middleware layer on top of Solaris. Both Dionisys and the quality event mechanism are currently being implemented in the Linux kernel. Dionisys utilizes CPU and network service managers, that support runtime service adaptation. In the Solaris implementation, network service management utilizes the DWCS packet scheduling algorithm [23], while CPU service management utilizes the real-time scheduling capabilities of the operating system. The Solaris operating system enables CPUs to be dedicated to user processes that run in the real-time priority class. Consequently, we have built a CPU service manager as part of Dionisys, using the *priocntl()* system call, that controls the priority and time-slice of processes competing for CPU cycles. For the Linux kernel implementation of Dionisys, the CPU service manager will employ a variant of DWCS for controlling the allocation of CPU cycles amongst competing processes and threads.

Figure 1 depicts a number of application-specific service managers, complemented by two more 'generic' service managers that control how CPU and network services are delivered to application processes. For example, for the video server described later in the paper, an application-specific service manager could control frame resolution, while the CPU and network service managers control the rate at which frames are generated and transmitted across a network, respectively. Quality events, transported via event channels between these service managers, cause frame resolution and rate to be adapted in keeping with the video server's quality of service requirements.

The implementation of quality events in Dionisys is based on the Data Exchange library [5] and uses two queues in each service manager: one queue for application-specific monitor functions and another for application-specific handler functions. Each service manager is responsible for executing its monitors and handlers at appropriate times: monitors are executed at application-specific times and handlers are executed in response to events generated by monitors. A service manager runs when it must either execute its service policy, execute one or more monitors, execute one or more handlers in response to pending events, or process requests from applications to create new event channels.

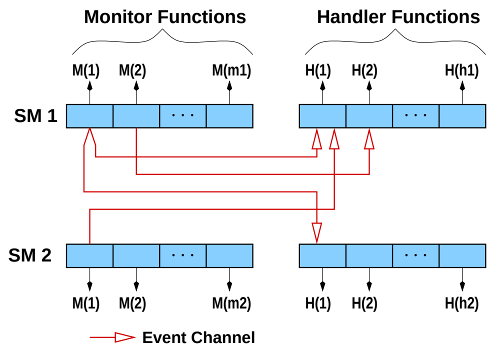

Figure 2. Example event channels involving two service managers, $SM1$ and $SM2$.

Figure 2 shows an example of several event channels between two service managers, $SM1$ and $SM2$, in Dionisys. Each service manager, SMi, maintains an array of pointers to monitor functions ($M(1)$ to $M(mi)$), and an array of pointers to handler functions ($H(1)$ to $H(hi)$). Furthermore, each service manager executes at most one monitor function and one handler function per application process. Consequently, there is at most one handler for an event channel in any one service manager, but there can be multiple handlers, spanning different service managers, connected to the same event channel. Each event channel binds one or more handler functions to a single monitor. For example, Figure 2 shows two handlers connected via the same event channel to monitor $M(1)$, in service manager $SM1$.

In the Solaris-based implementation of Dionisys, service managers on the same host all execute as kernel-level threads, within the same address space [1]. Application processes communicate with service managers on the same host via shared memory, using calls to library routines offered by the Dionisys application-programming interface (API) [23]. The Dionisys API allows applications to create, or delete, application-specific service managers, and/or event channels, or to exchange data with existing service managers. The rationale for this implementation is to emulate the way in which operating system services are accessed via system calls from application processes. Furthermore, applications can specialize the operation of Dionisys service managers, or create new service managers, in the same way that kernel-loadable modules can be dynamically-linked into an 'extensible' operating system[2]. Stated concisely, when an application creates an application-specific service manager (which executes application-specific policies), the service manager functionality is compiled into a shared object and dynamically linked into the Dionisys system-level address space. For the user-level realization of Dionisys used in this paper, this address space is that of a daemon process running all Dionisys service managers. For the kernel-level realization of Dionisys now being developed by our group, this address space is that of the (Linux) operating system kernel. In this fashion, we can incorporate application-specific service managers into Dionisys, supporting configurable protocols and other functions applicable to, and provided by, each application. Finally, when running Dionisys across a distributed system, each host has its own Dionisys daemon process. A simple name-server, running on a single, known host, uniquely identifies each service manager executing within each and every daemon process. Sockets are then used to communicate application data, QoS attributes and events between these processes on different hosts.

Flexibility of Quality Management. As stated earlier, applications specify the functionality to monitor service quality. Monitor functions are executed at times determined by applications[3], and influence *when* adaptations occur. An event is typically raised when actual and required service levels differ by an amount that causes an application to jeopardize its QoS requirements. Alternatively, a monitor might generate an event if it is possible that *better* quality of service can be achieved by adapting one or more service types. *How* an event is handled actually depends on the handler function that is specified by the application, and executed by the target service manager. Furthermore, applications have the ability to control *where* monitor and handler functions are executed. That is, each application using the Dionisys QoS infrastructure has the ability to specify the service managers responsible for executing its monitor and handlers functions. In Dionisys, service managers dynamically-link with shared objects containing the monitoring and handling functions at runtime.

Observe that the issues of *protection* associated with dynamically linking application-specific monitoring, handling and service manager functionality into Dionisys are out of the scope of this paper. In fact, protection issues are currently being addressed in our kernel-level implementation of Dionisys and quality events.

Having briefly described the quality event mechanism and a specific example of its implementation in Dionisys, we now show how quality events can be used to construct an adaptable real-time, client-server application.

2.1. An Adaptable Client-Server Application

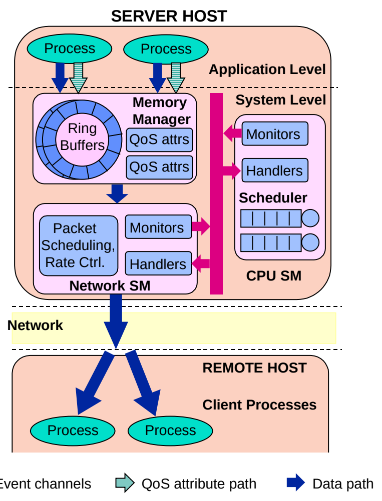

Figure 3. An example of an adaptable client-server application.

Figure 3 shows an adaptable client-server application, with event channels between CPU and network service

[1]It should be noted that the quality event mechanism, fundamental to Dionisys, imposes no restriction on how service managers are implemented in general.

[2]For example, Linux and Solaris both allow objects to be dynamically-linked into the kernel using 'insmod' and 'modload', respectively.

[3]In the Solaris implementation of Dionisys, there are limitations on the maximum rates at which services can be monitored due to service manager threads running in $10mS$ time-slices.

managers on the server host. Consider the case where Figure 3 represents a video-server. The CPU service manager uses a scheduling policy to allocate CPU cycles to server processes, that generate sequences of video frames. These video frames are split into packets and placed in ring buffers in shared memory, using an application-specific memory manager, that allocates one ring buffer for each video stream. Associated with each stream are QoS attributes, that specify parameters such as throughput and frame loss-rate. If a ring buffer is backlogged with too many packets, the network service manager can be configured to drop queued packets or discard incoming video frames. Furthermore, the network manager multiplexes packets over the outgoing network link, using a suitable packet scheduling policy. Once the video frames have been received at the destination, client processes decode and play back these frames.

We now discuss several different adaptation strategies that can be implemented using quality events in Dionisys, to control the allocation of resources to applications like the one just illustrated. In our cooperation with Honeywell Corporation, we have also developed similar adaptation strategies and experimented with them on an Automatic Target Recognition (ATR) application. Consequently, the adaptation strategies described in the remainder of this paper are applicable to many 'information flow' (or pipelined [10]) applications, and not just the one described in this section.

3. Adaptation Configurations

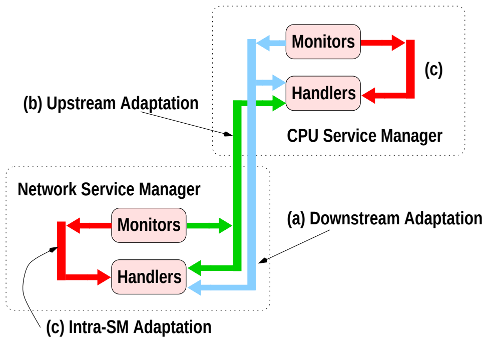

Figure 4. Adaptation configurations.

Figure 4 depicts three different adaptation configurations considered in this paper. For simplicity, we focus on adaptations between CPU and network service managers, but what follows applies to any combination of service managers:

- **Downstream Adaptation** – In this configuration, events are delivered from monitors in the CPU service manager to handlers within both the CPU and network service managers. Intuitively, events from the CPU service manager to the network service manager follow the forward-going path of application data, as it is generated and then transmitted.
- **Upstream Adaptation** – In this configuration, events are delivered from monitors in the network service manager to handlers within both the network and CPU service managers. The term 'upstream adaptation' is used to identify the situation where events are delivered in a direction *opposing* the logical flow of application data. This is the case for events from the network service manager to the CPU service manager.
- **Intra-SM Adaptation** – In this configuration, events are delivered from monitors to handlers *within* the corresponding service manager. Events are not exchanged between service managers, so there is no explicit *coordination* of multiple service types. A similar adaptation strategy has been developed by other researchers for real-rate scheduling [21], to self-adapt service managers that operate in pairs, to produce and consume information at a given rate.

Astute readers may notice that passing events between monitors and handlers is similar to a closed-loop feedback control mechanism, in which the handlers are like control *actuators*. All three adaptation configurations can be thought of as using feedback control loops, constructed using the quality event mechanism.

To emphasize the importance of both upstream and downstream adaptation strategies, consider the example of an *error control mechanism* used in a data communication system. In this example, the error control mechanism incorporates both forward-error correction (FEC) and retransmission techniques. Suppose that data is typically retransmitted if it is lost or received in error at the destination. In this case, a service manager, SM_{dest}, on the destination host executes a monitor function to determine whether data has been received correctly or not. If the data is received in error (or is lost), SM_{dest} sends a retransmission request, in the form of an event, back to a service manager, SM_{source}, on the source host. The request is handled by SM_{source} and the data is retransmitted. Observe that events acting as retransmission requests are delivered *upstream* with respect to the flow of data.

Now suppose a monitor in SM_{source} observes that certain data is frequently being retransmitted. In this case, it may be beneficial to apply a forward-error correcting code to this class of data before it is transmitted, thereby eliminating retransmission delays at the cost of using extra CPU cycles to encode and decode the data. With forward-error correction, SM_{source} sends an event *downstream*, and in synchrony with the encoded data, to a handler in SM_{dest} that knows how to decode the data.

If forward-error correction requires more CPU cycles than are available at a given point in time (at the source and/or destination), the communication system might decide to switch back to (or continue to use) a retransmission policy. The decision depends on the cost of each error control technique to an application.

The above example illustrates a situation where both downstream and upstream adaptation strategies are useful. The flexibility of quality events, allows both types of adaptation strategies to be implemented, as well as others. The next section evaluates several adaptation strategies using a simple video server application, like the one described in Section 2.1.

4. Experimental Evaluation

We ran a series of experiments, using a pair of Solaris-based SparcStation Ultra II machines connected via Ethernet, to show trade-offs in different service adaptation strategies. These strategies differed in *how*, *when* and *where* service adaptations took place. A video server was constructed as in Section 2.1. Server processes generated video frames, while client processes decoded and played back frames arriving from the network. The CPU service manager used a static priority preemptive scheduling policy, to schedule all application processes (one per video stream), while the network service manager scheduled all packets of video frames from the heads of shared memory ring buffers (again, one ring buffer per stream), using dynamic window-constrained scheduling (DWCS) [23]. Since the network service manager was implemented as a Solaris kernel thread, in a Dionisys daemon process, the packet scheduler did not access the Ethernet device directly, but via sockets.

In the first set of experiments, 1000 MPEG-1 I-frames[4] from a 'Star Wars' sequence were generated for three different streams, s_1, s_2 and s_3. The QoS attributes associated with each stream were minimum, maximum and target frame rates. The QoS attributes for s_1, s_2 and s_3 were 10 ± 2, 20 ± 2 and 30 ± 3 frames per second (fps), respectively. A QoS violation occurred when the actual transmission rate was out of the specified range for the corresponding stream, at the time it was monitored by the network service manager.

In these experiments, the following adaptation configurations were compared:

- **Downstream Adaptation.** The CPU service manager monitored each stream's frame *generation* rate: if the actual generation rate was not equal to the target rate, an event was sent to the CPU service manager and (forward to) the network service manager.
- **Upstream Adaptation.** The network service manager monitored each stream's frame *transmission* rate: if actual transmission rate was not equal to the target rate, an event was sent to the network service manager and (back to) the CPU service manager.
- **Intra-SM Adaptation.** Both CPU and network service managers independently monitored each stream's generation and transmission rates, respectively, and sent events to themselves if a stream's target rate was not equal to its actual rate.

In all cases, the monitor functions ran every $T = 10$ milliseconds, to ensure that QoS state information was sampled fast enough. The CPU service manager executed the same handler for each stream. The handler attempted to alter the Solaris real-time priority of the corresponding stream-generator process by an amount that was a *linear* function of the difference in target and actual service rates. That is, the priority, $prio_i$, of stream s_i's generator process, changed by an amount $\delta prio_i = K_i(\rho_{i,target} - \rho_i(nT))$, where K_i is some proportionality constant, $\rho_i(nT)$ is the nth sampled value of service rate, and $\rho_{i,target}$ is the target service rate. In the first set of experiments, $K_i = K, \ \forall i$, where K is a constant and $1 \leq i \leq m$ if there are m streams in total. A more elaborate control scheme could employ a PID [5] function of target and actual service rates.

To ensure one stream was not unduly serviced at the cost of other streams, the CPU service manager executed a *guard function*, which ensured priority assignments were within a valid range. Policing in the CPU service manager was used to prevent the highest-priority process from generating frames at a rate greater than the maximum required generation rate. This forced the CPU scheduler to run as a *non-work-conserving* scheduler.

The network service manager ran the same handler function for all three streams. The handler function adapted DWCS scheduling parameters to alter the allocation of service in a specific window of time. That is, each stream s_i was given $\frac{y_i - x_i}{y_i}$ fraction of network bandwidth every window of y_i time units, where x_i and y_i were tunable parameters. Consequently, we were able to control the number of consecutive packets of video frames transmitted from the same stream in a given time interval. Finally, policing in the network service manager was used to prevent the actual rate of a stream from exceeding its maximum transmission rate.

Buffering and Missed Deadlines. Figure 5 shows upstream adaptation has greater variance in buffer usage. The results are shown for s_1, but the same observations apply to all streams, irrespective of their target rate. The maximum number of backlogged frames for a stream, and the periods in which the ring buffers are empty, is less with both downstream and intra-SM adaptations, than with upstream adaptation. Observe that with upstream adaptation, the network monitor functions raise events too late, to request the generation of more frames. That is, control events are generated when nothing can be done until *future* data is generated. By contrast, downstream adaptation can often lead to adaptation changes in synchrony with the *current* data, as

[4] 160x120 pixels per frame, and 24 bits per pixel.

[5] 'PID' means 'proportional plus integral plus derivative'.

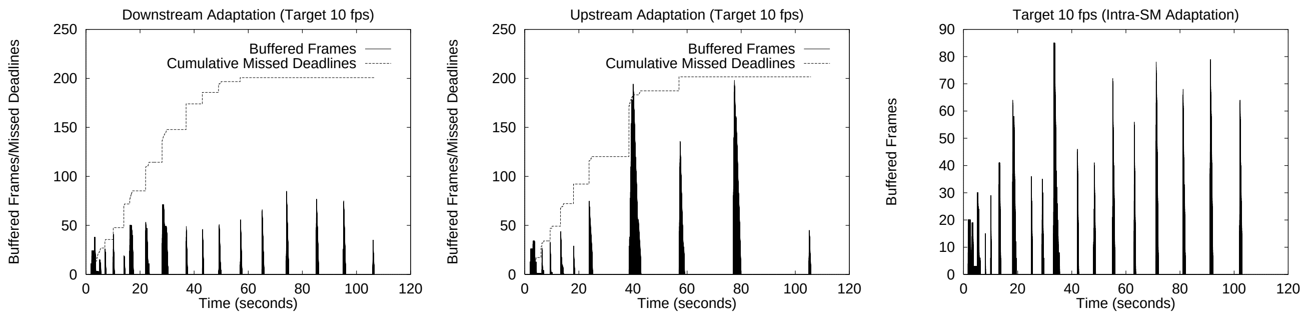

Figure 5. Buffering of the 10fps stream, s_1, with (a) downstream, (b) upstream, and (c) intra-SM adaptation. Cumulative missed deadlines are also shown for downstream and upstream adaptations.

it traverses the logical path of resources, to its destination. As a consequence, Figures 5(a) and (b) show that downstream adaptation causes fewer *consecutive* missed deadlines than upstream adaptation, when the deadlines on consecutive frames of s_1 are $100mS$ apart.

The latency of sending events from the network service manager to the CPU service manager, with upstream adaptation, causes buffers to be empty for relatively long periods. Likewise, buffers can continue to fill before events are generated to reduce the fill rate. In our experiments, the latency of an event sent (from a monitor to a handler) within the same service manager thread was $101.3\mu S$, while it was as high as $10.2mS$ when sent between service manager threads. Inter-thread event handling is more expensive than intra-thread event handling, due to the time-slice of a thread being fixed by the operating system at $10mS$. Observe that with downstream and intra-SM adaptation, we send events from monitors to handlers within the CPU service manager thread. Consequently, CPU cycles are allocated with finer-grained control in downstream and intra-SM adaptation than upstream adaptation.

QoS Range Violations. Figure 6(a) shows that there are most QoS range violations with upstream adaptation. However, intra-SM adaptation is also worse than downstream adaptation. This is because there is no *coordination* between CPU and network service managers. Consequently, intra-SM adaptation requires more time than downstream adaptation to reach the steady-state, in which service rate is within range of the maximum and minimum allowed values.

Mean Queueing Delay. The mean queueing delay levels off over time using downstream adaptation but continues to change significantly with upstream adaptation (as shown in Figure 6(b), for s_3). With upstream adaptation, the sudden increases in delay occur when each stream's queue length (ring buffer fill-level) increases. Furthermore, greater mismatches between service rates at the CPU and network-levels lead to more variation in the number of buffered frames, and more variation in mean queueing delay.

Finally, the lack of coordination between service managers, using intra-SM adaptation, is also responsible for the rise in mean delay of s_3 over time. However, intra-SM adaptation is appropriate for applications that can tolerate the lack of 'explicit' coordination between service managers. Moreover, intra-SM adaptation does not suffer from the cost of communicating events between service managers.

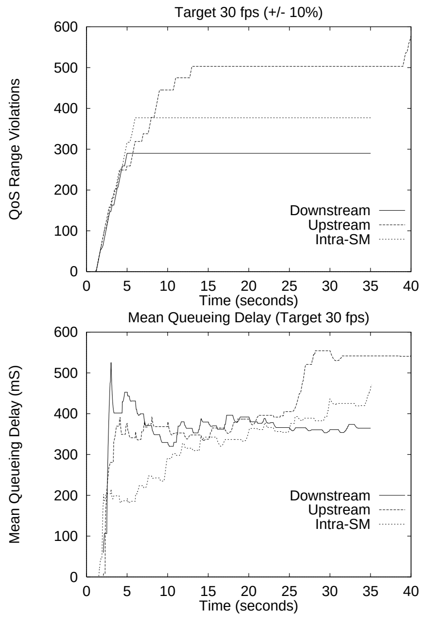

Figure 6. (a) Cumulative QoS range violations, and (b) mean queueing delay versus time for buffered frames, of the 30fps stream, s_3.

4.1. Quality Functions

We have shown how quality events can be used to implement different runtime service adaptation strategies, all of which are suitable for different situations. With quality events, it is also possible to embed 'quality functions' into monitors and handlers, thereby influencing how resources are best allocated to applications. The goal of such quality functions is to maximize total quality of service across multiple applications [1], using formulations of service quality appropriate to each application. We now show how such quality functions can be utilized to improve overall service quality to all applications, competing for common resources.

In this set of experiments, frame-generator processes generated 1000 frames for stream s_1, 2000 frames for s_2 and 3000 frames for s_3. The service rates of s_1, s_2 and s_3 were now set to 10 ± 1, 20 ± 2 and 30 ± 3 frames per second, respectively. Furthermore, to model the effect of dynamically-changing resource availability, s_2 and s_3 were blocked for exponentially-distributed idle times, averaging 3 seconds, after generating each set of 1000 frames.

The same monitoring and handling functions, as in the previous set of experiments, were used here. As before, the CPU service manager executed a handler function that attempted to alter the Solaris real-time priority of the corresponding stream-generator process by an amount that was a *linear* function of the difference in target and actual service rates. However, the extent to which a Solaris priority was altered was a function of the stream's own quality function. Such quality functions are similar to the idea of reward [1], value [12], or payoff functions [14].

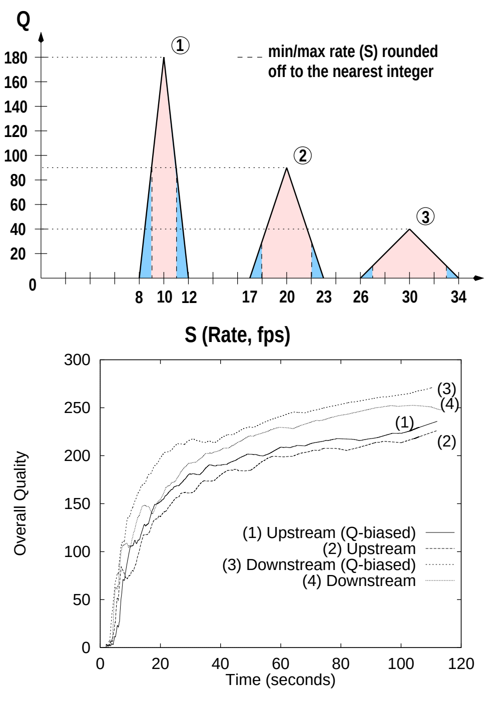

Figure 7. (a) QoS functions for each of the three streams, and (b) overall quality versus time.

The actual quality functions that were used are shown in Figure 7(a). These functions show the quality, Q, as perceived by each stream, for the corresponding service rate, S. For example, the priority of s_1 is adjusted by a factor of 3 more than the priority of s_2 (based on the ratio of the gradients of the quality functions for s_1 and s_2). In essence, this implies that s_1 is 3 times more critical than the s_2, when the actual and target service rates of both streams differ by the same amount. Likewise, s_2 is three times more 'quality critical' than s_3. These quality functions can be incorporated into monitors and/or handlers of different managers, assuming there exists a functional translation [6, 10] from the service offered by a given service manager to application-perceived quality. Exactly what a given quality means to an application is application-specific. For example, the quality Q may be a unit-less quantity, signifying some level of satisfaction as seen by an application user, or it may be translated to some specific units denoting the value to an application of a given quality of service.

Figure 7(b) shows how overall quality is improved, using different adaptation strategies, when CPU service is biased to the more quality critical stream. The 'Q-biasing' graphs refer to the results when the quality functions shown in Figure 7(a) are used. By contrast, all other graphs show results when CPU service is adjusted *equally* for competing streams, even though one stream's service is really more critical than another's. The overall quality at time t, $Q_{overall}(t)$, was calculated from the sum of the average sampled quality of each stream. That is, $Q_{overall}(t) = \sum_{i=1}^{m}(\frac{\sum_{j=1}^{n_{s_i}(t)} Q_j(s_i)}{n_{s_i}(t)})$, where m is the number of streams, $n_{s_i(t)}$ is the number of times the service quality of s_i is monitored (sampled) by time t, and $Q_j(s_i)$ is the jth sampled value of Q for stream s_i. In these experiments, the optimal value of $Q_{overall}(t)$ is 310, which is the sum of the peak values of Q for each stream from Figure 7(a). The overall quality using downstream adaptation and Q-biasing approaches 87% of the optimal value, when $t = 110$ seconds.

Figures 8(a) and (b) show network service rates for all three streams, using downstream and upstream adaptations, respectively. As can be seen, s_1 spends less time away from its target rate than both s_2 and s_3, because s_1 is more quality critical. Better service is provided to the more important streams, in order to improve overall quality. Also shown in Figure 8, as a series of grey bands, are the acceptable service (and, hence, QoS) regions. Ideally, all three streams should have service rates as close as possible to the middle of the QoS regions, based on the quality functions shown in Figure 7(a). It is important to note that both upstream and downstream adaptations maintain acceptable service rates for all three streams most of the time, and each adaptation method attempts to keep the service rates on target as often as possible.

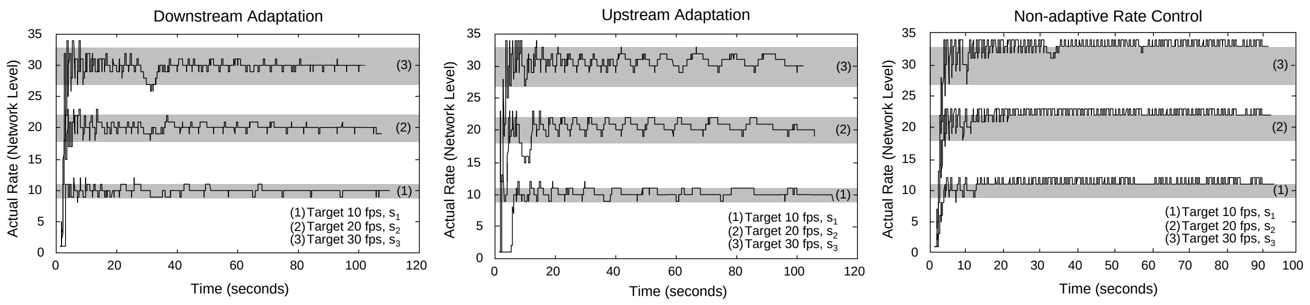

Figure 8. Network service rate, with (a) downstream adaptation and Q-biasing, (b) upstream adaptation and Q-biasing, and (c) non-adaptive rate control.

By comparison to Figures 8(a) and (b), Figure 8(c) shows the network service rate of all three streams when non-adaptive rate control is used. In the non-adaptive method, CPU time was divided into slots so that s_3 received 3 time slots, s_2 received 2 time slots, and s_1 received 1 time slot every window of 6 time slots. That is, CPU time was reserved for each stream based on the corresponding target rate requirement. Each time slot was $10mS$, thereby guaranteeing, for example, s_3 received $30mS$ of CPU time every $60mS$. At the network-level, the packet scheduler scheduled packets of video frames for transmission at rates proportional to the target service rates of each stream. This situation meant that sufficient processing capacity and network bandwidth was reserved to meet the maximum service rates of all three streams, but non-adaptive rate control was used to prevent resources being used beyond the amounts needed to meet those maximum service rates. When a stream's generation or transmission rate exceeded the maximum rate, rate control prevented that stream from using anymore of its reserved resources. Non-adaptive rate control was also used with the upstream and downstream adaptation methods, but they also adapted the allocation of resources to try to ensure each stream was serviced as close as possible to its target rate. In contrast, non-adaptive rate control made no attempt to guarantee each stream was serviced at its target rate.

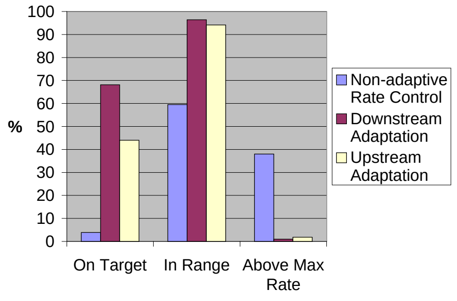

Figure 9. Overall performance of adaptive and non-adaptive rate control methods.

In the steady state, non-adaptive rate control results in the smallest peak-to-peak service oscillations. However, the oscillations have the highest frequency and the service rate is rarely on target. This is evident from Figure 9, which shows the percentage of time that all three streams were collectively transmitted: (a) at their target rates, (b) within their ranges, and (c) above their maximum rates, when both adaptive and non-adaptive rate control methods were used.

5. Conclusions and Future Work

This paper describes *quality events*, a software mechanism using which an application and/or system can be extended to enable runtime QoS management. Such extensions offer flexibility in: (1) `how` application- and/or system-level services are dynamically managed to maintain required quality, (2) `when` such management occurs, and (3) `where` this service management is performed. As part of the quality event mechanism, application-specific *monitor* and *handler* functions are associated with service providers, to perform service monitoring and management tasks.

To demonstrate the importance of *flexible* quality management via quality events, several adaptation strategies are compared in this paper: 'upstream adaptation', 'downstream adaptation' and 'intra-SM adaptation'. Upstream adaptation can lead to poor quality control if adaptation latencies are significant. This is analogous to the problem associated with feedback congestion control[11], in that, if the time to inform the producer that it should reduce its generation rate, is greater than the maximum time available to effectively apply such an adaptation, then the consumer can be flooded with too much information. With downstream adaptation, the delay between generating a control event and enacting the necessary service change can be coupled with the time to transfer application data along the logical path between service managers. That is, downstream adaptation events can be sent in synchrony with the flow of data. However, downstream adaptation is not always possible, especially when service adaptations cannot be enacted in a target service manager to compensate for service inadequacies earlier in the logical path along which information flows. For example, a target service manager may not be able to allocate enough CPU cycles to decode data encoded by a forward error correction method. By contrast, intra-SM adaptation works well if: (1) *coordination* between resources and, hence, service managers is not important, or (2) there is access to shared state information (e.g., buffer fill levels), which allows groups of service managers to coordinate their own service adaptations.

Quality events also allow 'quality functions' to be embedded into monitors and handlers, thereby permitting applications to influence how resources are best allocated to maximize their own notions of quality of service. Experimental results show how such quality functions can be utilized to trade-off service quality across multiple applications that compete for common resources. These results also demonstrate that adaptive resource management strategies can lead to more efficient use of resources, and better qualities of service for certain 'information flow' applications than non-adaptive resource management methods.

Finally, future work involves adding features to quality events that perform stability and error analysis, as well as 'QoS-safety' checks on application-specific handlers, thereby ensuring that service changes do not adversely affect the behavior of the QoS management system.

References

[1] T. Abdelzaher and K. Shin. End-host architecture for QoS-adaptive communication. In *The 4th IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[2] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems Journal, Special Issue on QoS Architecture*, 1997.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, and B. E. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *The 15th ACM Symposium on Operating Systems Principles*, December 1995.

[4] DARPA Quorum project list: http://www.darpa.mil/ito/research/quorum/projects.html.

[5] G. Eisenhauer, B. Schroeder, K. Schwan, V. Martin, and J. Vetter. Data exchange: High performance communication in distributed laboratories. In *The 9th International Conference on Parallel and Distributed Computing and Systems (PDCS'97)*, October 1997.

[6] D. Ferrari. Client requirements for real-time communication services. *IEEE Communications Magazine*, 28(11):76–90, November 1990.

[7] D. Ferrari and D. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal of Selected Areas in Communications (JSAC)*, 8(3):368–77, 1990.

[8] A. Goel, D. Steere, C. Pu, and J. Walpole. SWiFT: A feedback control and dynamic reconfiguration toolkit. Technical Report 98-009, OGI CSE, 1998.

[9] J. Huang, Y. Wang, N. Vaidyanathan, and F. Cao. GRMS: A global resource management system for distributed QoS and criticality support. In *IEEE International Conference on Multimedia Systems and Computing*. IEEE, June 1997.

[10] D. Hull, A. Shankar, K. Nahrstedt, and J. W. S. Liu. An end-to-end QoS model and management architecture. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, pages 82–89, December 1997.

[11] V. Jacobson. Congestion avoidance and control. In *ACM Computer Communication Review: Proceedings of the SIGCOMM '88*, pages 314–329. ACM, August 1988.

[12] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *IEEE Real-Time Systems Symposium*. IEEE, December 1985.

[13] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *The 16th ACM Symposium on Operating System Principles*, December 1997.

[14] R. Kravets, K. Calvert, and K. Schwan. Payoff-based communication adaptation based on network service availability. In *IEEE Multimedia Systems'98*. IEEE, 1998.

[15] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal of Selected Areas in Communication*, 17(9):1632–1650, September 1999.

[16] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reservation for multimedia operating systems. In *IEEE International Conference on Multimedia Computing and Systems*. IEEE, May 1994.

[17] K. Nahrstedt and J. Smith. The QoS broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[18] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. Practical solutions for QoS-based resource allocation problems. In *The 19th IEEE Real-Time Systems Symposium*, December 1998.

[19] D. Rosu, K. Schwan, and S. Yalamanchili. FARA - a framework for adaptive resource allocation in complex real-time systems. In *The 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*, Denver, USA, June 1998.

[20] D. I. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On adaptive resource allocation for complex real-time applications. In *The 18th IEEE Real-Time Systems Symposium, San Francisco, USA*, December 1997.

[21] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *The 3rd Symposium on Operating System Design and Implementation*. USENIX, 1999.

[22] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, September 1998.

[23] R. West. *Adaptive Real-Time Management of Communication and Computation Resources*. PhD thesis, Georgia Institute of Technology, August 2000.